

On the Implementation of a Modified Sag-Szekeres Quadrature Method*

J. N. Lyness

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439 U.S.A.

L. M. Delves

Institute of Advanced Scientific Computation
University of Liverpool L69 3BX
England

Abstract

We describe a modified Sag-Szekeres multidimensional quadrature algorithm and discuss its implementation as a general-purpose library procedure on serial and parallel architectures. Examples illustrate its effectiveness for both smooth and singular integrands.

1 Introduction

In the practice of numerical quadrature, many different special-purpose algorithms are available and are efficient when used in the appropriate circumstances. We are interested in developing an algorithm for more general use as a software item in a software library. Specifically we seek a general purpose algorithm that accepts as wide a class of integrands and regions as possible, without grave compromise of efficiency. To this end, we have chosen one suitable for N -dimensional integration over a product region

$$\mathcal{R}_N = \mathcal{R}_1^1 \times \mathcal{R}_1^2 \times \dots \mathcal{R}_1^N,$$

where each \mathcal{R}_1^j stands for one of $[a, b]$, $[a, \infty)$, $(-\infty, b]$ or $(-\infty, \infty)$. The algorithm that we describe uses a modified version of the Sag-Szekeres (1964) method in each direction. It is efficient when the integrand function is regular over \mathcal{R}_N or when it has integrable singular behavior confined to vertices or edges of this region. This ability to handle edge singularities without special coding makes it particularly appealing for a numerical library.

*This work was supported in part by the Mathematical, Information, and Computational Sciences Division sub-program of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and by the Commission of the European Community within Esprit Project P2528: Supernode II. lyness@mcs.anl.gov; delves@liverpool.ac.uk
March 1998

2 One-Dimensional Algorithm for the Finite Interval [0,1]

As a preliminary to constructing a general N -dimensional algorithm, we focus on one dimension and on the finite interval $[0,1]$. Here we seek an algorithm that will handle an integrable singularity at $x = 0$ or at $x = 1$. The appropriate Gaussian rule is undoubtedly the most efficient rule known. However, to implement this requires that a weight function incorporate the singularity; weights and abscissas depend on the actual weight function. Extrapolation quadrature is marginally less efficient and marginally more general than the Gaussian rule; it requires only limited information about the singularity. Yet even this information, we believe, may not be normally available to the applications programmer.

However, a modification of the Sag-Szekeres approach does seem to be promising. Here the trapezoidal rule is used, but it is applied to a transformed function. The transformation is or may be the same for all integrands. The resulting rule is efficient for integrands that are analytic in $(0,1)$ and integrable in $[0,1]$.

Following Sag-Szekeres (1964), we set

$$\begin{aligned}\psi(t) &= \frac{1}{2}(1 + \tanh(1/(1-t) - 1/t)), \\ \psi'(t) &= \frac{1}{2}((1-t)^{-2} + t^{-2})(1 - \tanh^2(1/(1-t) - 1/t)),\end{aligned}\tag{2.1}$$

and set $x = \psi(t)$ to effect

$$If = \int_0^1 f(x)dx = \int_0^1 f(\psi(t))\psi'(t)dt.$$

We denote the integrand on the right by $F(t)$. We now apply an m -panel trapezoidal rule, namely,

$$Q^{(m)}F = \frac{1}{m} \sum_{j=0}^m {}''F(j/m) = \frac{1}{m} \sum_{j=0}^m {}''f(\psi(\frac{j}{m}))\psi'(j/m),$$

to approximate

$$IF = \int_0^1 F(t)dt = \int_0^1 f(\psi(t))\psi'(t)dt = If.$$

The expression $Q^{(m)}F$ may be treated as a conventional quadrature rule for $f(x)$ whose abscissas are $\psi(j/m)$ and whose weights are $\psi'(j/m)$. Many functions $\psi(t)$ of form (2.1) are suitable; the one chosen above was used by Sag and Szekeres (1964). This rule was later used by Murota and Iri (1982), who called it the TANH rule and noted that it was a variant of the IMT method of Iri, Moriguti, and Takasawa (1970) (see also Davies and Rabinowitz (1980)). (Takahasi and Mori (1973) use the term TANH to refer to a completely different transformation.)

Several other choices for $\psi(t)$ are described in the literature. All necessarily satisfy

$$\psi(0) = 0, \quad \psi(1) = 1 \text{ and } \psi'(t) > 0 \text{ for } t \text{ in } (0,1),$$

and all suggested to date have $\psi'(t)$ symmetric about $t = 1/2$. Perhaps the earliest is a set due to Korobov (1963); the most recent is a set due to Sidi (1993). For these, the functions $\psi'(t)$ comprise

a set of algebraic polynomials and a set of trigonometric polynomials, respectively. For integrands without singularities, undoubtedly Sidi's functions are excellent and probably more efficient than the ones we have proposed. We have retained the choice (2.1) because in some respects it is more convenient for functions with unbounded singularities at the end point.

Clearly when $f(x)$ is bounded in a finite interval (λ, μ) and $\psi(t)$ is monotonic and differentiable, it follows that $F(t) = f(\psi(t))\psi'(t)$ is also bounded in this interval. When $f(x)$ has a singularity at $x = 0$, it may or may not happen that $F(t)$ has a singularity at $t = 0$. This depends on the natures of $f(x)$ and of $\psi(t)$. To fix ideas, let us suppose $f(x) = x^\alpha$. Then we have

$$F(t) = \psi(t)^\alpha \psi'(t) = \frac{1}{\alpha + 1} \frac{d}{dt} \psi(t)^{\alpha+1} \quad \alpha \neq -1,$$

and we may confirm that $F(t)$ is integrable

$$IF = \int_0^1 F(t) dt = \frac{1}{\alpha + 1} \psi(t)^{\alpha+1} \Big|_0^1 = \frac{1}{\alpha + 1} \quad \alpha \neq -1$$

and is naturally identical with If . For some values of α , it may happen that $F(t)$ has a singularity at $t = 0$.

Theorem 2.1 *For $f(x) = x^\alpha$ and choice (2.1) for $\psi(t)$, we have*

$$F(t) \sim (1/t^2)(\exp(-2/t))^{\alpha+1} \text{ as } t \text{ approaches } 0+.$$

This is straightforward to prove.

It follows that, even though $f(x)$ is singular at $x = 0$, the natural continuation of $F(t)$ and all its derivatives at $t = 0$ are zero. Clearly one may omit the function value at $t = 0$ in forming the trapezoidal rule sum. Ultimately, the convergence rate of a sequence of trapezoidal rule approximations is exponential in the number of panels used.

In this case, the corresponding functions of Sidi, while robust, do not produce a sequence that converges exponentially. This situation is illustrated by the circumstance that for α greater than but sufficiently close to -1 , the limit in the theorem is infinite when $F(t)$ is calculated using these trigonometric polynomials for $\psi'(t)$.

3 The Numerical Stability of the One-Dimensional Algorithm for Interval [0,1]

The formulas given in Section 2 appear to be straightforward to implement. In several distinct places, however, careful programming is required to avoid unnecessary inaccuracy or breakdown resulting from unexpected overflow or inconvenient underflow. Some of the underlying causes for sensitivity are interrelated. All are connected with function evaluation at or near the integration interval endpoints. (In the multidimensional extension considered later, this stability problem occurs in each dimension separately.)

In the following discussion, it is important to distinguish between the underflow parameter, ϵ_u , and the machine accuracy parameter, ϵ_m . We shall illustrate the discussion by setting $\epsilon_u = 10^{-73}$ and $\epsilon_m = 10^{-12}$. This discussion is in the context of a machine with quiet underflow. That is, left to itself, any number too small to be represented is simply replaced by zero.

The density of machine-representable numbers plays a key role in quantifying, understanding, and controlling the numerical instability. Naturally, one tries to arrange the calculation so that the most sensitive calculations are carried out where this density is greatest, namely near the origin.

In general, the smallest positive machine-representable number is the underflow parameter, ϵ_u . Between ϵ_u and $2\epsilon_u$ are $1/\epsilon_m$ different machine-representable numbers. In general, when X is a power of 2, there are $1/\epsilon_m$ machine-representable numbers regularly arranged between X and $2X$. This pattern continues until the largest machine-representable number (usually approximately or exactly $1/\epsilon_u$). The negative machine-representable numbers follow almost exactly the same pattern.

As mentioned earlier, we treat the interval $[0,1]$. We term the zero end of this interval the “sensitive” end, since there we can distinguish numbers very close to each other, this distance being of order ϵ_u . We term the other end the “insensitive” end. The corresponding distance here is ϵ_m . To help control the calculational error, we introduce the quantities $\bar{x} = 1-x$ and $\bar{\psi}(t) = 1-\psi(t)$. It turns out that for t in the interval $(0.95,1)$, the nearest machine-representable number to $\psi(t)$ is 1. In some cases we can organize the internal coding so that we use $\bar{\psi}(t)$ and avoid $\psi(t)$. This allows a more sensitive calculation. But for t in $(0.99,1)$, we find $\bar{\psi}(t)$ is represented by zero in the machine. The end-point problem is mitigated but not removed.

It is a straightforward exercise to program the calculation of $\psi(t)$, $\bar{\psi}(t) = 1 - \psi(t)$, and $\psi'(t)$ so that each is available to near machine accuracy. Only one exponential call is required to obtain all three. As mentioned above, when $t \leq 0.01$ and when $t \geq 0.99$, either $\psi(t)$ or $\bar{\psi}(t)$ is smaller than ϵ_u and hence cannot be represented in the machine. Normally, such a number would be replaced by zero. For reasons that will become apparent later, we recommend that these minute numbers be replaced by ϵ_u . However, when appropriate, we happily allow $\psi'(t)$ to be replaced by zero. When $t \leq 0.05$ or $t \geq 0.95$, either $\psi(t)$ or $\bar{\psi}(t)$ is less than ϵ_m . Note that all these quantities, however small, are properly calculated to machine accuracy — except, of course, when they are too small to be represented.

The calculation involves the numerical integration, using the trapezoidal rule, of the integrand function $F(t) = f(\psi(t))\psi'(t)$. Since $\psi'(t) = 0$ at the endpoints, it is obvious that when $f(x)$ is bounded, the endpoint contribution is zero and can be omitted. As shown above, when $f(x)$ has an integrable singularity at $x = 0$ or $x = 1$, the integrand function $f(\psi(t))\psi'(t)$ is zero at $t = 0$ and at $t = 1$. At these values of x , function evaluation of $f(x)$ is unnecessary.

In theory, the abscissa $x = \psi(t)$ is 0 or 1 only when $t = 0$ or 1; otherwise, $x = \psi(t)$ in $(0,1)$. So, in an ideal world where there is “infinite-precision arithmetic”, we can safely use the trapezoidal rule to approximate the integral, simply ignoring the two endpoint function values. In practice, however, values of $\psi(t)$ may appear that are closer to 0 (or 1) than to any other machine-representable number. It is necessary to ensure that, in such cases, these are not replaced in the machine by 0 (or 1). If that were to happen and $f(x)$ happens to have a singularity there, an overflow would occur.

To obviate this possibility, the quadrature routine should replace $\psi(t)$ by $\max\{\psi(t), \epsilon_u\}$ near $t = 0$ and by $\min\{\psi(t), 1 - \epsilon_m\}$ near $t = 1$. Then it will not ask for a function value of $f(x)$ precisely at an endpoint of its integration interval. Naturally, the used-provided procedure from which $f(x)$ is calculated, must not overflow for any machine-representable number $x \in (0, 1)$. It is important to emphasize this precaution because, while function values at $x = 0$ and $x = 1$ are not required, function values at points x exceptionally close to $x = 0$ or $x = 1$ may well be required.

Corresponding restrictions should be applied independently to $\bar{\psi}(t)$. However, $\psi'(t)$ should not be restrained in this way. When this is too small to be represented, it is replaced by zero.

The above remarks cover the situation at $t = 0$ and $t = 1$. Next we turn to the situation near these endpoints. To clarify our ideas, we look at the trapezoidal rule sum

$$\frac{1}{m} \sum_{j=1}^{m-1} [f(\psi(j/m))\psi'(j/m)].$$

For integrands $f(x)$ that are regular, one may be tempted to omit terms for which $\psi'(t)$ is less than the machine accuracy parameter. Doing this, one omits about 5% of the integration interval at each end. If the program omits the corresponding function evaluation, a 10% economy may ensue. However, in some cases unnecessary inaccuracy could arise: for example, if $f(x)$ were large very near an endpoint but minuscule elsewhere. In particular, there is no justification for this doubtful economy when $f(x)$ has any sort of singular behavior at either endpoint.

To illustrate these remarks, we look at three examples, namely, $f(x) = 1$, $f(x) = x^{-2/3}$, and $f(x) = (1 - x)^{-2/3}$. The exact integrals If are 1, 3, and 3, respectively. We consider the fifty-panel trapezoidal rule sum ($m = 50$). In the first two examples we examine the contribution of the three terms $j = 1, 2, 3$ to this trapezoidal rule sum. This is

$$\begin{aligned} &0.02 * (f(\psi(0.02)) * \psi'(0.02)) + f(\psi(0.04)) * \psi'(0.04) + f(\psi(0.06)) * \psi'(0.06)) = \\ &0.02 * (0.14\text{D} - 38 * f(0.28\text{D} - 42) + 0.19\text{D} - 17 * f(0.15\text{D} - 20) + 0.15\text{D} - 10 * f(0.28\text{D} - 13)). \end{aligned}$$

We write this as

$$w_1 f_1 + w_2 f_2 + w_3 f_3$$

with

$$w_j = 0.28\text{D} - 40, \quad 0.38\text{D} - 19, \text{ and } 0.30\text{D} - 12, \quad (3.1)$$

respectively.

In the first example, $f(x) = 1$, and the first three terms contribute precisely these amounts to a sum that is approximately 1. The first two terms, which are comfortably smaller than ϵ_m , have no practical effect on the result. If all three and the corresponding three at the other end of the interval are ignored, the result may be compromised by an amount 0.60D-12.

That was a particularly simple example. The second example, $f(x) = x^{-2/3}$, is quite different. The three function values involved are not all 1, but they are large. They are

$$f_j = 0.23\text{D} + 29, \quad 0.786\text{D} + 14, \text{ and } 0.11\text{D} + 10, \quad (3.2)$$

respectively. Their respective contributions to the sum are obtained by multiplying them by the weights in (3.1) above, giving

$$w_j f_j = 0.64D - 12, \quad 0.29D - 5, \text{ and } 0.33D - 3, \quad (3.3)$$

respectively. Terms of this size cannot be routinely omitted simply because one of the factors involved in their evaluation is small. Note that the computer has all these numbers available to machine accuracy (i.e., in this example, to twelve decimal places). To make this description easier to read, we have written down only the first two places in the above discussion.

The third example, $f(x) = (1-x)^{-2/3}$, is again different. Because of symmetry, one might have expected this example to correspond in all significant respects to the previous example. However, because the singularity is at the end $t = 1$, the situation is much worse. Here the critical points are the final three. We can calculate the weights correctly; these are the same as in (3.1) above. The correct function values f_{50-j} and the correct values of $w_{50-j} f_{50-j}$ are those in (3.2) and (3.3) above. But $\psi(t)$ has to be represented in the machine and has to be less than 1. The nearest machine representable number is $x = 1 - \epsilon_m$, and so the largest value of $f(x)$ calculable is about $1.00D+8$. Hence, instead of function values (3.2) we find all three to be about $1.00D+8$, which are much too small. The true contributions (3.3) to the overall sum are then underestimated, leaving an overall error of about $0.30D-3$.

Clearly, a singularity at $t = 1$ is unwelcome. The user should, if possible, arrange that the singularity occurs at the $t = 0$ end of this integration interval, possibly reprogramming the integrand function to exploit the higher density of machine- representable numbers in that neighborhood. However, as we shall see later, the user need not worry about any singularity induced by the transformation from an infinite or semi-infinite interval. This is taken care of automatically in any proper implementation.

4 The One-Dimensional Algorithm for Other Intervals

One advantage of our approach is that it can be modified to intervals other than $[0,1]$ by means of an additional transformation of a user-provided function $g(y)$. This transformation, denoted by $y = \alpha(x)$, is chosen so that

$$\int_a^b g(y) dy = \int_0^1 f(x) dx.$$

Here we allow either or both of a and b to be infinite but assume, when germane, that $b > a$. Naturally there is a wide choice of possible transformations. For our program, we have chosen

$$\begin{aligned} y &= a + (b-a)x, & f(x) &= (b-a)g(a + (b-a)x), & [a, b] \\ y &= a + \frac{1-x}{x}, & f(x) &= x^{-2}g(a + (1-x)/x), & [a, \infty) \\ y &= b - \frac{1-x}{x}, & f(x) &= x^{-2}g(b - (1-x)/x), & (-\infty, b] \\ y &= \frac{1}{1-x} - \frac{1}{x}, & f(x) &= (x^{-2} + (1-x)^{-2}) \\ & & & g(1/(1-x) - 1/x) & (-\infty, \infty). \end{aligned} \quad (4.1)$$

In our implementation, the user provides $g(y)$. It is clear from the transformations that when $|g(y)|$ is bounded in (a, b) , then $|f(x)|$ is bounded in $(0, 1)$. But it is easy to show that when a or b is finite, then any singularity of $g(y)$ at a or b may induce a corresponding singularity of $f(x)$ at 0 or 1. Moreover, in general, when the interval is semi-finite or doubly infinite, one may encounter a transformation-induced singularity in $f(x)$ at the end of $[0, 1]$ which corresponds to infinity.

In the preceding section, we discussed in some detail the care necessary to deal with a singularity in $f(x)$. In the present case, the user provides $g(y)$ and our program determines $f(x)$. We have to arrange that this part of our program provides an integrand function $f(x)$ that is finite for all machine-representable $x \in (0, 1)$. Doing so is not difficult because the terms that induce the singularity are $x = \psi(t)$ or $\bar{x} = 1 - x = \bar{\psi}(t)$, and these can be determined so long as x or \bar{x} exceeds ϵ_u .

For example, on the semi-finite interval $[a, \infty)$, suppose

$$g(y) = (\lambda^2 + (y - a)^2)^{\alpha/2}$$

with $\alpha < -1$ to ensure convergence. This gives rise to

$$\begin{aligned} f(x) &= x^{-2} g\left(a + \frac{1-x}{x}\right), \\ &= x^{-2} x^{-\alpha} (\lambda^2 x^2 + (1-x)^2)^{\alpha/2}. \end{aligned}$$

For noninteger α , this has a singularity at $x = 0$. In the integration of $f(x)$, the quantity x^{-2} is critical. This quantity is isolated by the program and accurately calculated. The value of $g\left(a + \frac{1-x}{x}\right)$ for x close to zero is small but is readily calculable and not sensitive to small changes in y . Thus, the coding of this can safely be left to the user. When the singularity is at the insensitive end $x = 1$, the term \bar{x} is provided by the program and plays the same role as above.

We note that the program for the finite interval demands that, for all machine-representable numbers in $(0, 1)$, the function $f(x)$ not exceed the highest machine-representable number. To ensure this, the user must provide a function $g(y)$ that does *not* produce overflow in $f(x)$ when $f(x)$ is calculated using one of (4.1).

The user may exploit the result in the following theorem by choosing M near the overflow parameter and “capping” the integrand function $g(y)$ appropriately.

Theorem 4.2 *Let $g(y)$ satisfy*

- | | | |
|--------------------------|---|-------|
| a, b finite | $ g(y) < M/(b - a)$ for all y , | (i) |
| a finite, b infinite | $ g(y) < M/4$ and $(y - a)^2 g(y) < M/4$ for all y , | (ii) |
| both infinite | $ g(y) < M/12$ and $y^2 g(y) < M/5$ for all y . | (iii) |

Then $|f(x)| < M$ for all $x \in (0, 1)$.

Proof. Part (iii) may be established as follows. The fourth transformation in (4.1) yields

$$f(x) = \left(y^2 + \frac{2}{(1-x)x} \right) g(y) := F_0 g(y), \quad (4.2)$$

where, using $y(1-x)x = 2x - 1$ we may express $F_0 = F_0(x)$ as a function of x .

- (a) When $x \in \left[\frac{1}{3}, \frac{2}{3} \right]$, $F_0(x)$ is a convex function symmetric about $x = \frac{1}{2}$; its minimum in this interval is $F_0(\frac{1}{2}) = 8$ and its maximum $F_0(\frac{1}{3}) = F_0(\frac{2}{3}) = 11\frac{1}{4}$. Using this (4.2) and the first inequality in hypothesis (iii) above, we find

$$|f(x)| = F_0 g(y) < M \quad \text{for all } x \in \left[\frac{1}{3}, \frac{2}{3} \right].$$

- (b) When $x \in (0, \frac{1}{3})$, we exploit the second inequality in hypothesis (iii) in much the same way. We set

$$f(x) = F_2 y^2 g(y)$$

and find $F_2(x) = F_0/y^2$ to be monotonic increasing in this interval, with $F_2(0) = 1$ and $F_2(\frac{1}{3}) = 5$. Thus,

$$|f(x)| = F_2 y^2 |g(y)| < M \quad \text{for all } x \in (0, \frac{1}{3}).$$

The same result for $x \in (\frac{2}{3}, 1)$ establishes part (iii) of the theorem.

Parts (i) and (ii), which are simpler, are established in a similar way.

5 The Multidimensional Algorithm

The extension of the algorithm to more than one dimension is trivial: we use a product trapezoidal rule with product mapping. There is, however, additional interest in the implementation details, and we discuss these in the context of a MIMD distributed-memory architecture.

The sums required are product trapezoidal rule sums. In the context of a parallel computer, one convenient method for evaluating any product sum is using a cyclic distribution of the function evaluations. We describe this now in a four dimensional setting in a slightly more general context than we need. The generalization to other dimensions is straightforward.

We consider a product rule of the form

$$QF = \sum_{j_1=1}^{n_1} \sum_{j_2=1}^{n_2} \sum_{j_3=1}^{n_3} \sum_{j_4=1}^{n_4} F(x_{j_1}^1, x_{j_2}^2, x_{j_3}^3, x_{j_4}^4) w_{j_1}^1 w_{j_2}^2 w_{j_3}^3 w_{j_4}^4. \quad (5.1)$$

In our application we ignore boundary points so $n_i = m_i - 1$ and, in each dimension, all weights are equal so

$$w_{j_i}^k = 1/m_k \quad j_i = 1, 2, \dots, n_i; \quad k = 1, 2, 3, 4.$$

We now reindex this sum, using a single index ℓ defined by

$$\begin{aligned}\ell &= j_1 + n_1 j_2 + n_1 n_2 j_3 + n_1 n_2 n_3 j_4 \\ &\equiv j_1 + n_1(j_2 + n_2(j_3 + n_3 j_4)).\end{aligned}$$

It is straightforward to verify that this mapping is one to one and that $\ell \in [1, L]$ with $L = n_1 n_2 n_3 n_4$. Given a value of $\ell \in [1, L]$, one may find j_1, j_2, j_3 , and j_4 by successive division. The sum (5.1) may be reexpressed, first in the form

$$QF = \sum_{\ell=1}^L F(\mathbf{x}_\ell) w(\mathbf{x}_\ell), \quad (5.2)$$

and then, with any integer $p \geq 1$, in the form

$$QF = \sum_{q=1}^p S_q = \sum_{q=1}^p \left(\sum_{\substack{\ell \equiv q \pmod{p} \\ \ell \in [1, L]}} F(\mathbf{x}_\ell) w(\mathbf{x}_\ell) \right). \quad (5.3)$$

The overall effect is that we have partitioned the sum in (5.1) into p different and distinct sums, which may be handled respectively by the p different processors. The number of elements in each sum S_q is either $\lfloor L/p \rfloor$ or $\lfloor L/p \rfloor + 1$.

The interesting aspect of a program to effect this is that there is no need for any processor to be explicitly aware of the values of ℓ involved. All of the processors are initially provided with (or calculate simultaneously) a list of weights and abscissas $x_{j_k}^k, w_{j_k}^k$ $j_k = 1, 2, \dots, n_k$ $k = 1, 2, 3, 4$. Each processor handles a selection of *allowable* indices (j_1, j_2, j_3, j_4) , that is, a set where each j_i is *within limit*, namely, $j_i \in [1, n_i]$.

The program handles an allowable index (j_1, j_2, j_3, j_4) by adding into a running sum the contribution

$$w_{j_1}^1 w_{j_2}^2 w_{j_3}^3 w_{j_4}^4 F(x_{j_1}^1, x_{j_2}^2, x_{j_3}^3, x_{j_4}^4).$$

The q -th processor is initialized by being given index $(q, 1, 1, 1)$. (As long as $q \in [1, n_1]$, this is allowable. If it is not, one applies the procedure described below to transform this index into an allowable index.)

After an allowable index (j_1, j_2, j_3, j_4) has been processed, the next index considered is $(j_1 + p, j_2, j_3, j_4)$. If this is allowable, it is processed immediately. Otherwise, it is transformed into an allowable index by applying a sequence of transformations, each of the type

$$T_i \begin{cases} j_i = j_i - n_i \\ j_{i+1} = j_{i+1} + 1. \end{cases}$$

If j_1 is out of limits, transformation T_1 is applied as many times as necessary to put j_1 into limits. Next T_2 and then T_3 are applied in the same way. Should j_4 become out of limits (while j_1, j_2, j_3 are in limits), the part of the calculation assigned to this processor is complete. The same algorithm may be described in the following way.

- α) if $j_1 > n_1$, then $j_1 = j_1 - n_1$ and $j_2 = j_2 + 1$; goto α)
- β) if $j_2 > n_2$, then $j_2 = j_2 - n_2$ and $j_3 = j_3 + 1$; goto β)
- γ) if $j_3 > n_3$, then $j_3 = j_3 - n_3$ and $j_4 = j_4 + 1$; goto γ)

If it finds $j_4 > n_4$, the sum is complete and the processor should return its contribution to the first (or a master) processor or, in some other way, amalgamate the distinct sums.

A program arranged in this way has several “computing virtues”:

1. Simplicity: Each processor is given an identical program.
2. Adaptability: p, n_1, n_2, n_3 , etc., appear as simple parameters.
3. Low Interprocessor Communication: Communication is needed only at the start (to assign the initial point) and at the end (to assemble the final result).
4. Even Load Balancing: The points have been shared as evenly as possible. Each processor takes a fair share of easy and difficult regions.

We close this section with some remarks about load balancing. The key to even load balancing is the elimination of processor wait time. If all function evaluations take an identical time (and there are many problems in which this is the case), then arranging even load balancing reduces simply to seeing that each processor treats, as far as possible, the same number of points. The scheme described above does this, as would most properly constructed schemes.

When function evaluation times differ from point to point, a more interesting or challenging situation arises.

It is convenient to define a difficult (easy) point as one where the function takes a longer (shorter) time than average to evaluate. A difficult (easy) region is one that contains a preponderance of difficult (easy) points. This depends only on the integrand function. A simple example of an easy region might be an edge where one component required in the calculation of the function value happens to be identically zero. An example of a difficult region might be an edge where, exceptionally, a limiting process has to be simulated to evaluate the function. Note that this depends simply on the time required to make the function evaluation. This is quite distinct from the concept of difficult or easy regions in the context of adaptive quadrature. That depends on the smoothness of the integrand.

The circumstances required for even load balancing are slightly different in a MIMD environment, where the processors act independently, from the circumstances in a SIMD environment, where the processors act in lock step. To pinpoint the difference, let us suppose that the order in which the abscissas were treated was entirely random. In a MIMD environment this is desirable. With luck, each processor would receive the same mix of easy and difficult points, so each would have the same amount of work to do and each would finish at about the same time; during the process, none have been kept waiting. On the other hand, this random ordering could be one of

the worst possible for a SIMD environment. The difficult points would be randomized too, and each time slot would contain a mix of difficult and easy points. Thus a processor apparently lucky enough to be treating an easy point might well find that, when it has finished this point, it has to wait until all other processors, some of which may be contemporaneously treating difficult points, have also finished.

Clearly, what is needed for both the MIMD and the SIMD environments is that each processor is assigned roughly the same number of difficult points and the same number of easy points. However, in the SIMD environment, the ordering may be critical while in the MIMD environment, this ordering is immaterial.

On the other hand, hypothetically, a good situation for a SIMD environment might be one in which the points were treated strictly in order of difficulty. All processors go slowly when the difficult points are being treated and all speed up when they treat the easy ones.

We now return to the scheme described above and see how these different environments react to a situation in which there exist well defined easy and difficult regions but it is not known a priori where these are. First, we note that the points of local regions are dispersed among the different processors. This is precisely what is wanted in both MIMD and SIMD environments.

In addition another effect may be helpful in a SIMD environment. Specifically, points in the same locality are being treated to some extent at the same time. To wit, there are roughly $[N/p]$ sets of p points that are treated simultaneously. Approximately a proportion of $(n_1 - p + 1)/n_1$ of these sets comprise p adjacent points. The time taken for each set is the time taken by the slowest (which is the most difficult) member of that set. Thus, when the difficult points occur in well-defined local regions, there is a good chance that, to some extent, difficult points will be processed at the same time.

6 Numerical Examples

The procedure described above has been implemented as a parallel library routine, running on transputer-based systems, as part of Esprit project P2528: Supernode II; (see Plowman (1992)). This routine is scheduled to appear in the quadrature section of the Liverpool-NAG Transputer Software Library. We give here some results obtained using this routine, to demonstrate the rapid convergence obtained with both smooth and singular integrands, and to demonstrate the routine's effectiveness on a parallel MIMD architecture.

6.1 Examples

We consider the following two dimensional problems, taken from Plowman (1992):

$$\begin{array}{ll}
 1. \int_2^3 \int_1^\infty x^{-y} dx dy = \ln 2 & 2. \int_0^\infty \int_0^\infty \exp(-x^2 - y^2) dx dy = \frac{\pi}{4} \\
 3. \int_0^1 \int_0^1 \frac{x}{\sqrt{x^2 + y^2}} dx dy = \frac{1}{2} \left(\ln(\sqrt{2} + 1) + \sqrt{2} - 1 \right) & 4. \int_0^\infty \int_0^\infty \sqrt{x + y} \exp(-x - y) dx dy = \frac{3\sqrt{\pi}}{4}
 \end{array}$$

6.2 Convergence of the Method

The convergence obtained by this method, as the number of function evaluations is increased, is exponential in nature. This is illustrated in Table 1.

In each of these 24 examples, the same number of panels, m , was used in each of the two dimensions. Thus, the number of function evaluations required in each example is $(m - 1)^2$. These results were obtained with four processors using 64-bit IEEE arithmetic. Once the machine precision is approached, the actual error depends slightly on the number of processors. This effect is not limited to a parallel implementation. Naturally, the final figure or two in any result depends on the actual coding. The last column gives the time taken, in seconds, for Problem 4 on one processor. Timings for the other problems are similar to these.

Table 1. Absolute error for four problems

m	Problem 1	Problem 2	Problem 3	Problem 4	Time (P.4)
4	1.3E-01	1.5	1.6E-01	1.9	0.11
8	3.3E-04	3.0E-01	1.1E-03	7.4E-02	0.11
16	2.8E-07	2.8E-02	7.2E-06	7.5E-03	0.13
32	1.6E-10	1.9E-04	2.0E-08	5.8E-06	0.18
64	8.9E-16	4.0E-08	1.1E-11	2.2E-12	0.41
128	2.2E-16	4.4E-16	1.8E-15	2.2E-16	1.31

These results are demonstrably consistent with exponential convergence for Problems 1 and 3 and superexponential convergence for Problems 2 and 4.

6.3 Parallel Performance

It is no surprise that the method implements well on a parallel distributed-memory architecture, since multidimensional quadrature methods are in general “embarrassingly parallel”. In Table 2 we illustrate this by giving the measured speedup factors

$$S(p) = T(1)/T(p),$$

where $T(n)$ is the time required when using n processors. Times were taken on a Transtech T800 system with 20 MHz processors and links; the configuration uses a master/slave paradigm with T800 master. We give two sets of results, using $m = 64$ and $m = 128$, respectively. Absolute timings were given in Table 1.

Table 2. Speedup factors $S(p)$ for p slave processors

p	Problem 1	Problem 2	Problem 3	Problem 4
$m = 64$				
2	1.8	1.8	1.8	1.8
4	3.1	3.0	2.9	3.1
8	4.4	4.2	3.9	4.3
$m = 128$				
2	2.0	2.3	1.9	1.9
4	3.8	3.6	3.6	3.7
8	6.6	6.7	6.1	6.8

The results are as expected: for fixed m and increasing p , the speedup factor achieved is limited by the initialization of the library routine rather than by the cost of the final collection of partial sums from each processor. With the dynamic loading mechanism used for the transputer library, this initialization cost is dominated by the cost of sending the code for the integrand function to each slave, which takes place when the routine is called. Naturally, this cost is relatively less significant for larger problems. Indeed, the timings obtained provide an estimate of about 0.11 seconds for the overheads involved.

7 Acknowledgments

The routine used was implemented by Dr. Steve Plowman; we are indebted to him for many lengthy discussions of implementation details.

References

- [1] P. J. Davies and P. Rabinowitz (1980), *Methods of Numerical Integration*, 2nd edition, Academic Press, New York (1980), pp. 142–144.
- [2] N. M. Korobov (1963), *Number-Theoretic Methods of Approximate Analysis*, GIFL, Moscow (1963) (in Russian).
- [3] M. Mori (1978), An IMT-Type Double Exponential Formula for Numerical Integration, *Pub. Res. Inst. Math. Sci. Kyoto Univ.*, **14**, pp. 713–729.
- [4] M. Iri, S. Moriguti, and Y. Takasawa (1970), On a Certain Quadrature Formula, *Kokyuroku of the Res. Inst. for Math. Sci. Kyoto Univ.*, **91**, pp. 82–118 (in Japanese). English translation in *J. Comp. Appl. Math.*, **17**, pp. 3–20.
- [5] K. Murota and M. Iri (1982), Parameter Tuning and Repeated Application of the IMT-type Transformation in Numerical Quadrature, *Numer. Math.*, **3**, pp. 347–363.
- [6] S. Plowman (1992), *Trapezoidal Rule Quadrature Algorithms for MIMD Distributed Memory Computers: Part 2 Implementation*, Supernode II Working Paper, Centre for Mathematical Software Research, University of Liverpool, U.K. (1992).
- [7] T. W. Sag and G. Szekeres (1964), Numerical Evaluation of High-Dimensional Integrals, *Math. Comp.*, **18**, pp. 245–253.
- [8] H. Takahasi, and M. Mori (1973), Quadrature Formulas Obtained by Variable Transformation, *Numer. Math.*, **21**, pp. 206–219.
- [9] A. Sidi (1993), A New Variable Transformation for Numerical Integration, in H. Brass and G. Hämmerlin (editors) *Numerical Integration IV*, Birkhauser, Berlin; ISNM **112**, pp. 359,373.